# Office integration

## *Introduction to Office integration*

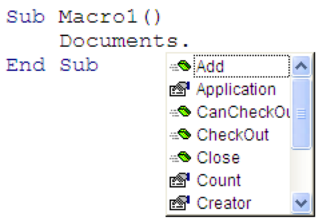
In this section we study how Office applications can communicate and integrate by using Visual Basic.

## *Integrating with Word*

Let's suppose we frequently summarise Excel information in a Microsoft Word document. Doing that is repetitive and time-consuming and could be better done by a Visual Basic macro. We'll look at how to do this. We begin with a simple example. Our macro, running in Excel, will start Word, create a document and then put text into the document. This example, and the following ones, will work on your PC only if you have Word installed.

### Type and Object libraries

Every Office application has a "type" or "object" library. The library contains a description of that application's object model. For example Excel's library describes the objects, properties and methods that apply to Excel. Word's library describes the objects, properties and methods that apply to Word. Visual Basic's Intellisense (the intelligent code editor) uses information in these libraries when it gives you its prompts. A snapshot of a typical Intellisense prompt is shown below.



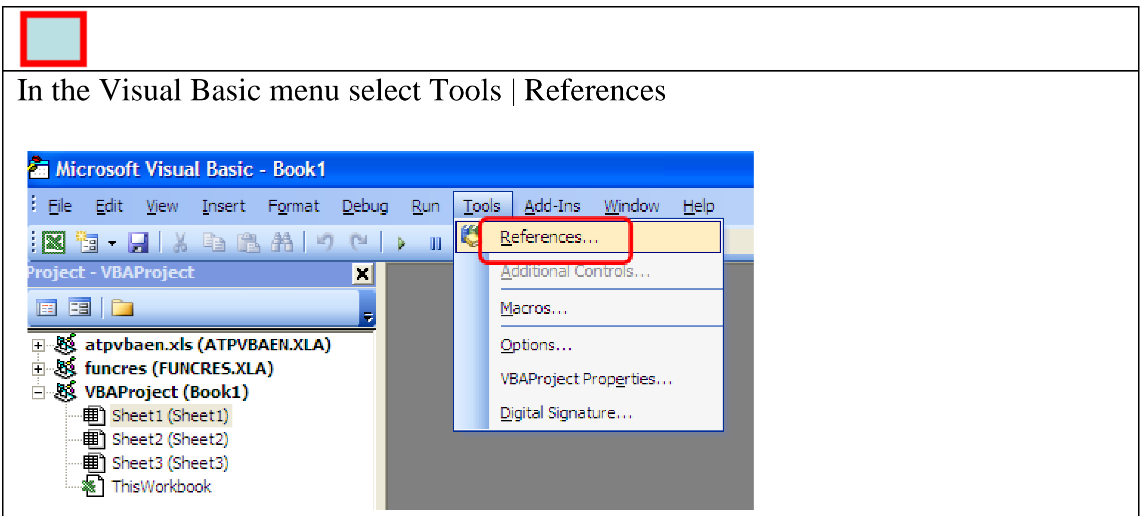

Here Intellisense is listing properties and methods that apply to a collection of Documents. You can, for example **Add** a Document to a collection. To list the prompts Intellisense needs information from a Word library.

Our Excel macro will need to refer to Word objects so we need Intellisense to "know" about Word. To do that we need to set up a reference to the Word library. That's what we'll do next. We begin by opening the Visual Basic editor.

| | |
|---|---|
| | • Open a new workbook<br>• In Excel select Tools \| Macro \| Visual Basic Editor |

Next we set up a reference to the Word library.

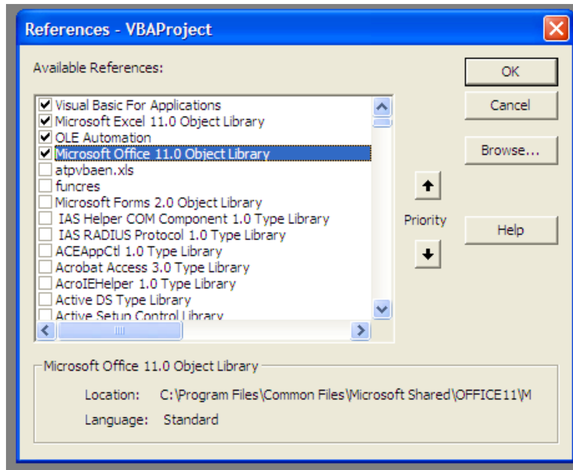
Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com



In the Visual Basic menu select Tools | References



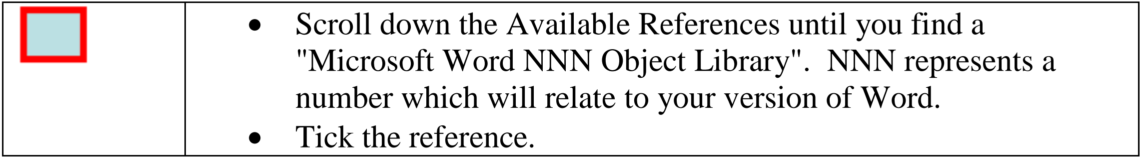
The list of available references will appear.





The active references are indicated by ticks. Unticked references are available to be used but aren't currently loaded.

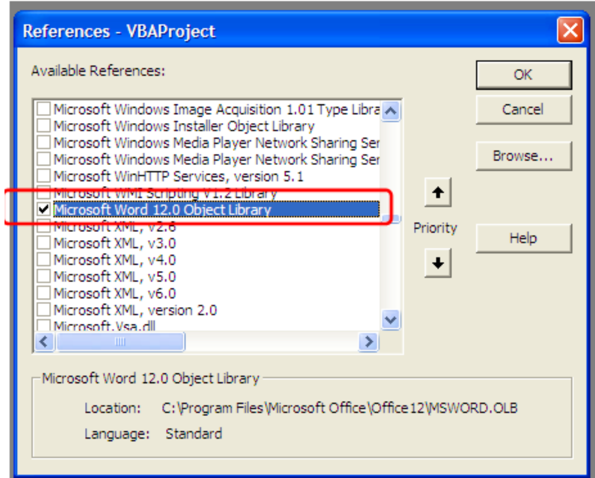
We search the list for a Word type library.

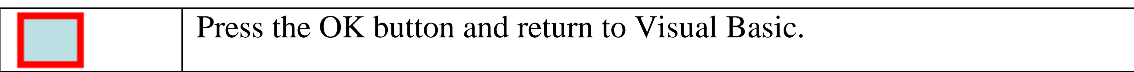


- Scroll down the Available References until you find a "Microsoft Word NNN Object Library". NNN represents a number which will relate to your version of Word.
- Tick the reference.

Your screen should look something like this.

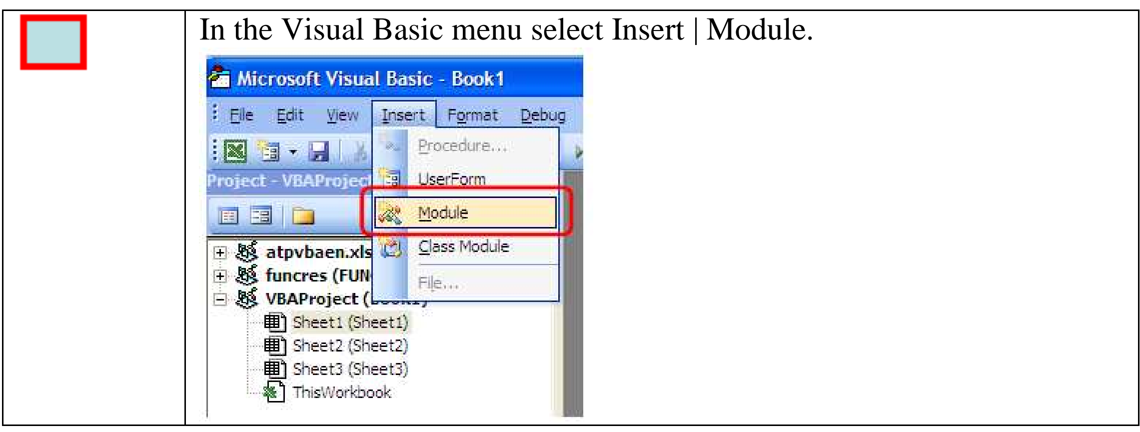
Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com



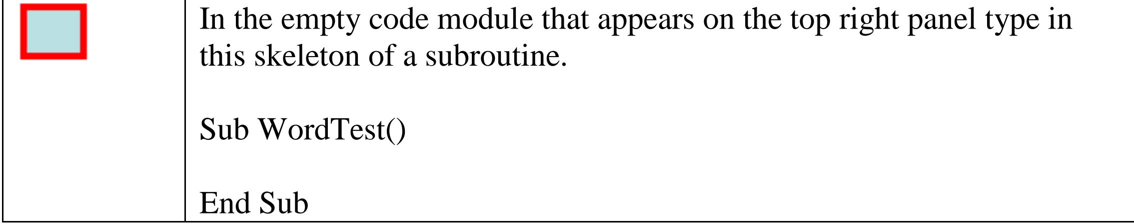
Press the OK button and return to Visual Basic.

Now Intellisense "knows" about Word and will give us prompts when dealing with Word objects. We can now begin writing Visual Basic code that refers to Word.



In the Visual Basic menu select Insert | Module.



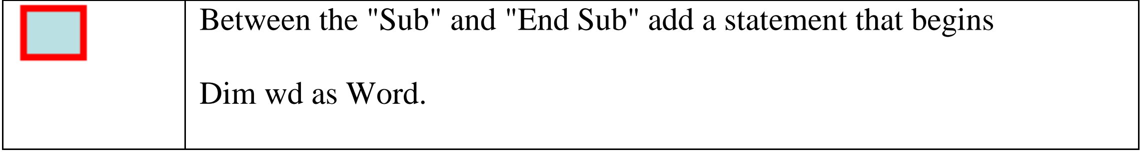


In the empty code module that appears on the top right panel type in this skeleton of a subroutine.

```
Sub WordTest()

End Sub
```

Next we design the body of the subroutine. We want to work with Word so let's define a variable that will later "point" to a Word application.



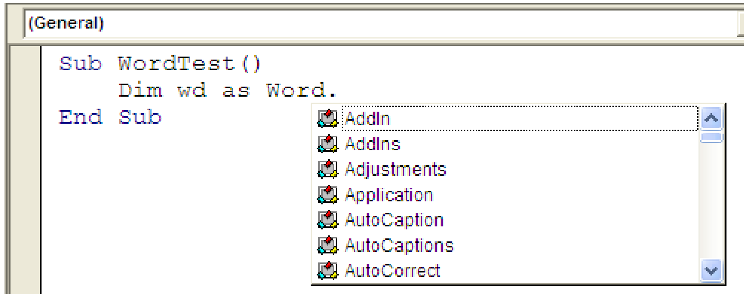
Between the "Sub" and "End Sub" add a statement that begins

Dim wd as Word.

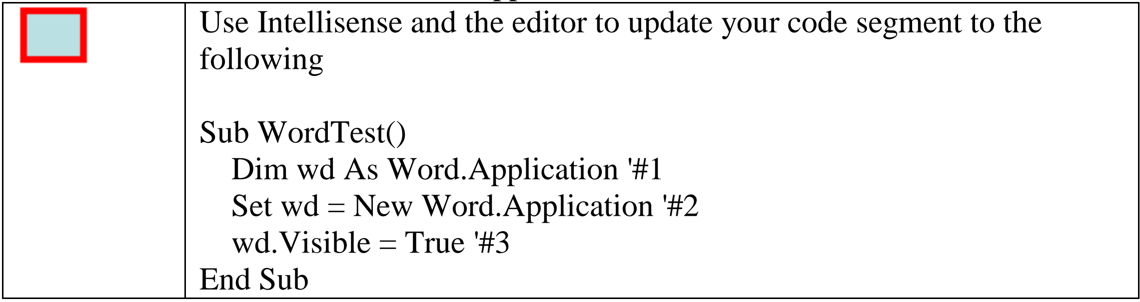
Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com

| | [Make sure to include the period after "Word"] |
|---|---|

Intellisense should the show you the properties and methods that apply to Word.




We will want to create a new Word application.



| | Use Intellisense and the editor to update your code segment to the following<br><br>Sub WordTest()<br>    Dim wd As Word.Application '#1<br>    Set wd = New Word.Application '#2<br>    wd.Visible = True '#3<br>End Sub |
|---|---|

In statement [#1] we define the variable wd to refer to a Word Application.
In [#2] we create the new Word application and we "point" wd at that application.
By default the new application will be invisible so [#3] makes it visible.

| | Run your macro by mouse-clicking on the first, "Sub", statement and pressing the F5 - or run - key. |
|---|---|

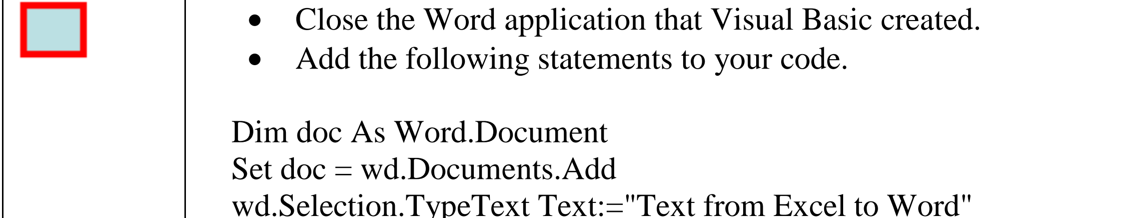
You should see something like the following on your screen.

```
Sub WordTest()
    Dim wd As Word.Application
    Set wd = New Word.Application
    wd.Visible = True
End Sub
```

Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com

An instance of the Word application has been created and made visible.

Next we'll add some code to make Visual Basic create a document within the application and to put text into the document.



- Close the Word application that Visual Basic created.
- Add the following statements to your code.

```
Dim doc As Word.Document
Set doc = wd.Documents.Add
wd.Selection.TypeText Text:="Text from Excel to Word"
```

Your code should now look like this.

```
Sub WordTest()
   Dim wd As Word.Application
   Set wd = New Word.Application
   wd.Visible = True
   Dim doc As Word.Document        '#1
   Set doc = wd.Documents.Add      '#2
   wd.Selection.TypeText Text:="Text from Excel to Word"        '#3
End Sub
```

In [#1] we define doc as a Word document.

In [#2] we point doc at a new document that we add to the collection (currently empty) in the wd application.
And in [#3] we use the TypeText method of the Selection property of the Word Application to add text to the document.

Run your macro.

An instance of Word should be created, a document added and text shown in the document.

Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com

```
Sub WordTest()
    Dim wd As Word.Application
    Set wd = New Word.Application
    wd.Visible = True
    Dim doc As Word.Document
    Set doc = wd.Documents.Add
    wd.Selection.TypeText Text:="Text from Excel to Word"
End Sub
```

That completes our first - small and simple - example.

## *Object libraries and portability*

Suppose we save our Excel workbook and mail it to someone else. Will our macro work for them? If they have the same version of the Word object library (12.0 - say) then the macro would work. But if they have a different version (11.0 - say) or if the file containing the library is in a different location then they'll get an error message.

```
Sub WordTest()
    Dim wd As Word.Application
    Set wd = New Word.Application
    wd.Visible = True
    Dim doc As Word.Document
    Set doc = wd.Documents.Add
    wd.Selection.TypeText Text:="Text from Excel to Word"
End Sub
```

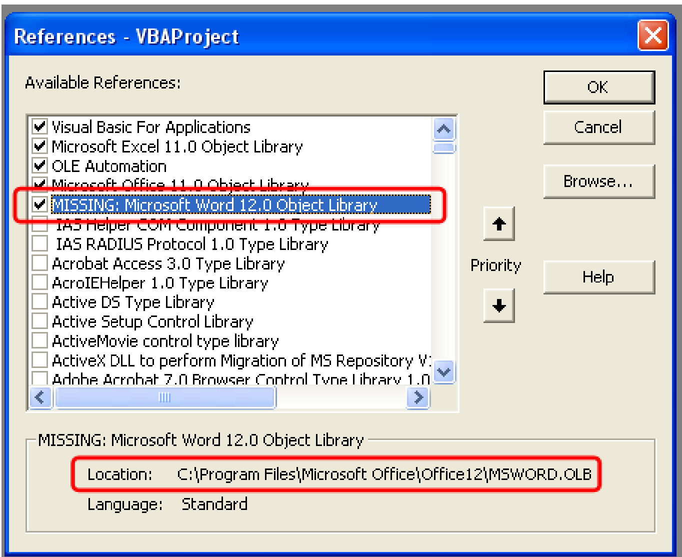
Microsoft Visual Basic

Compile error:

Can't find project or library

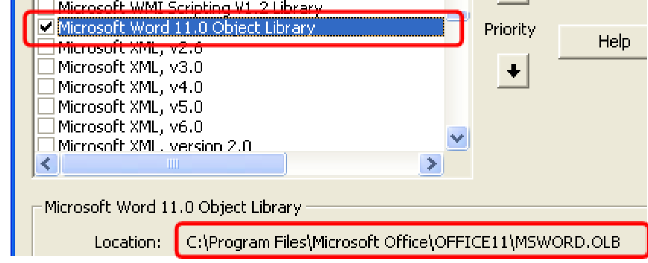
Visual Basic indicates it can't find the library. Why not? If you were to bring up the Tools | References dialog you'd see this.

Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com




The Microsoft Word 12.0 object library is missing. Why can't Visual Basic find it? For this reason: When you add a library reference in Visual Basic the spreadsheet you are using "remembers" the name and path of the library. And if the library has a different name or location on a recipient's PC then Visual Basic won't find the library. As a result any macros that reference the library will fail.

If this happens then spreadsheet's recipient can manually reset the reference to their own version of the library. In our example let's imagine the spreadsheet's recipient has their library at "C:\Program Files\Microsoft Office\OFFICE11\MSWORD.OLB". Then they can browse to that ..




.. and reset the reference.

Then the macro will work.

But going through this process is frustrating and time-consuming. Isthere a better way? Fortunately - yes. We'll see how in the next section.

## *Late binding versus early binding*

The macro we wrote earlier used a technique called early binding. Another technique called late binding results in macros that are easier to distribute. This is because late binding doesn't required explicit references to libraries.

Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com

We'll look now at late binding and see how it compares with early binding. First, we'll compare the late and early binding versions of the same macro.

| ***Early binding*** | ***Late binding*** |
|---|---|
| Sub WordTest()<br>Dim wd As Word.Application<br>Set wd = New Word.Application<br>wd.Visible = True<br>Dim doc As Word.Document<br>Set doc = wd.Documents.Add<br>wd.Selection.TypeText Text:="Text"<br>End Sub | Sub WordTest2()<br>Dim wd As Object<br>Set wd = CreateObject("Word.Application")<br>wd.Visible = True<br>Dim doc As Object<br>Set doc = wd.Documents.Add<br>wd.Selection.TypeText Text:="Text"<br>End Sub |

The differences between the two are highlighted in gray on the right hand side. You'll notice that the differences relate mainly to the type of objects in the Dim statements. With early binding we use exactly the type of object we need (e.g. a Word.Document or a Word.Application). In late binding we use the more generic "Object".

IntelliSense loses much of its usefulness when we switch to late binding. Let's see why.

| | |
|---|---|
| | Create a new macro with the following code. The macro is a late binding version of the first part of our earlier macro.<br><br>Sub WordTest2()<br>Dim wd As Object<br>Set wd = CreateObject("Word.Application")<br>End Sub |

| | |
|---|---|
| | Add the following after the Set statement and before the End Sub:<br><br>wd.<br><br>[Remember to add the period after the wd.] |

Your screen should look like this.

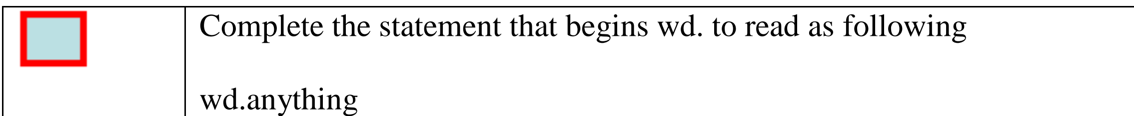
```
Sub WordTest2()
    Dim wd As Object
    Set wd = CreateObject("Word.Application")
    wd.
End Sub
```

No IntelliSense

Note there are no IntelliSense prompts. Because wd is now a generic object Visual Basic and Intellisense don't know what properties and methods apply to it. So it is

Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com

you - the code author - who needs to know the objects and methods that apply.  Visual Basic will simply "trust" that your objects and methods will make sense when the code runs.  At this stage you could type in anythin and Visual Basic would accept it.  We'll try.



Complete the statement that begins wd. to read as following

wd.anything

The code should now look like this:

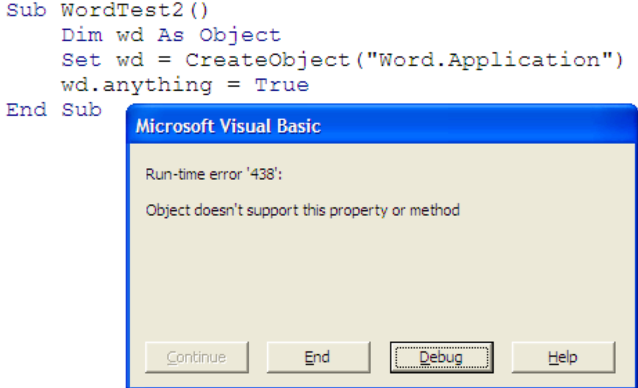
```
Sub WordTest2()
   Dim wd As Object
   Set wd = CreateObject("Word.Application")
   wd.anything = True
End Sub
```

Visual Basic should accept the code without complaining.

Next we'll run the code.  We should find that, only when Visual Basic runs the code, does it find a problem.

Run your macro

A run-time error will appear.




The behaviour we've seen sheds light on what late binding means: Treating objects as generic until they're actually needed at run-time.

We'll now fix our deliberate bug and test the late-binding version of the subroutine.

Update your code to be like this

Sub WordTest2()

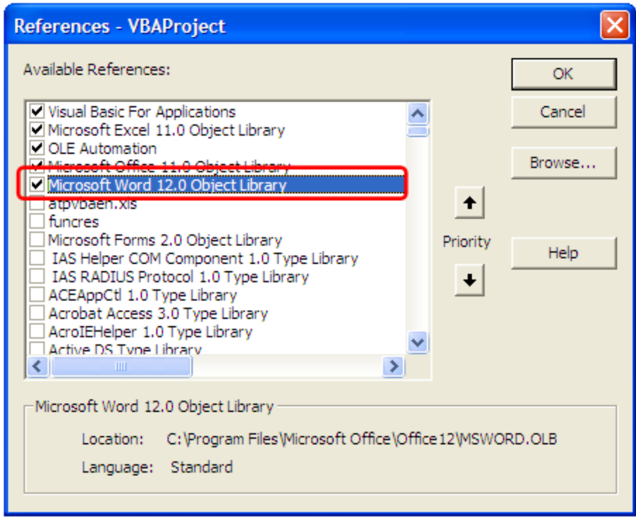
Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com

| | |
|---|---|
| | Dim wd As Object<br>Set wd = CreateObject("Word.Application")<br>wd.Visible = True<br>Dim doc As Object<br>Set doc = wd.Documents.Add<br>wd.Selection.TypeText Text:="Text"<br>End Sub |

| | |
|---|---|
| | Run the code and verify that it works as it should. |

To conclude this section let's test whether the code will continue to work if the Microsoft Word object library is removed from the set of active libraries.

| | |
|---|---|
| | Select Tools \| References |




| | |
|---|---|
| | • Un-tick the Microsoft Word Object Library.<br>• Press OK |
| | • Try running the early-binding version of your code. It should fail. |
| | • Try running the late-binding version of your code. It should continue to work. |

We have shown that late-binding removes the need to explicitly refer to particular Object libraries. That makes it easier to distribute spreadsheets and macros.

## Using early and late binding

Copyright © 2009 Tykoh Group Pty Limited
All rights reserved
www.tykoh.com